THE DEVELOPMENT OF A GENERIC SIGNING AVATAR

ABSTRACT
A signing avatar is an avatar that can perform sign language for the Deaf. We describe our design and implementation of a stand-alone generic signing avatar, which can be used for signing most sign languages and can be incorporated into other applications with minimal adaptation.

KEY WORDS
Avatars, animation, signing avatars, humanoid animation, H-Anim.

1 Introduction

Over the past decade, interest in sign language computer applications has increased substantially. Such applications include sign language dictionaries like the Auslan dictionary [1], teaching tools such as the SYNENOESSE project in Greece [2], and different machine translation systems from text to sign language [3, 4]. All these applications require a *signing avatar*; that is, an avatar that can execute signs from a supported sign language.

There are fundamental differences between normal avatars and signing avatars. Normal avatars such as those used in the movie and the gaming industry, typically require gross motor movement and computationally expensive collision avoidance with the environment. Signing avatars, on the other hand, require extremely fine motor movements and minimal collision avoidance routines [1]. In addition, a signing avatar is required to perform so-called non-manual signs. A non-manual sign comprises subtle facial movements, such as lifting of the eyebrows or a slight puffing of the cheeks. These non-manual signs are linguistically meaningful and essential components of any sign language.

Various signing avatars had been developed successfully in their encompassing environments, but all these avatars are either commercial products [7] or were integrated into their respective projects [6, 2, 3] to the extent that it is not easy to re-use them in other projects. In this article we describe our implementation of a non-commercial free generic signing avatar, which can be incorporated into other projects with minimal effort.

The rest of this article is organized as follows: in Section 2 we describe the issues surrounding signing avatars in general. We then present the design and implementation of our generic signing avatar in Section 3, illustrate our results in Section 4, consider future work in Section 5, and conclude in Section 6.

2 Background to Signing Avatars

All signing avatars must, in one way or another, address three distinct issues:

- the graphical model (that is, *appearance*) of the avatar,
- the *animations* applied to the model in order to show the signs, and
- the *notational interface* between the sign description in the dictionary, and the commands for the animation of the sign.

We discuss each of these three issues below.

Non-generic signing avatars typically feature only one avatar model, or otherwise allow only superficial differences in skin texture and facial appearance amongst their avatars. Realistic human avatars are time-consuming to construct, and as in movies and games, even the best avatars appear somewhat 'plastic' to observers. In addition, our user testing showed that users who use signing avatars in assistive technology tools for learning sign language, often have difficulty to pick up subtle facial expressions in human-like avatars. In order to build a generic signing avatar, we had to design a system that would work on any 'reasonable' avatar definition.

Animations are also typically restricted to the given avatar model. This means that, given an avatar system, the situation arises where the animations must be reworked on a programming level for each new avatar that is coupled to the system. In order to write generic animations, however, it is important that the avatar model conforms to a specific standard. The *de facto* standard for humanoid avatars is the H-Anim standard [8] and we also followed this standard. H-Anim compliant avatars are usually built using the Virtual Reality Modelling Language (VRML) or the XML encoding of VRML called X3D [9]. VRML models are displayed by embedding them into normal HTML web pages. To correctly view these web pages, they must be opened using a VRML compliant web browser such as Blaxxun Contact [10]. The animation capabilities of VRML is too simplistic for the complex animations needed in signing avatar systems, and the VRML External Authoring Interface (EAI) needs to be used to animate VRML signing avatars. However, Yang, Petrui and Whalen [11] showed that the animation frame rate suffers due to the overhead caused by the EAI. The other option for the display of VRML models is to convert the model to another suitable

[1]Signs in a sign language are mathematically well-defined: see Kennaway [5] on this issue.

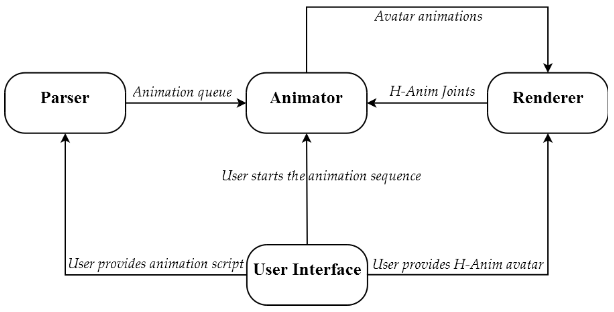

Avatar animations
Parser
Animation queue
Animator
H-Anim Joints
Renderer
User starts the animation sequence
User provides animation script
User Interface
User provides H-Anim avatar

Figure 1. The design of the generic signing avatar system

format and delegate the rendering of the model to the most convenient rendering mechanism for that format. This conversion is typically done using VRML file loaders, and suitable formats are formats that share the VRML hierarchical scene graph structure. In our case, we settled on the Java3D scene graph format [12].

The notational interface usually requires that the encompassing system can translate dictionary entries into animation sequences, and it is possible (albeit not always easy) to use standard notations such as XSTEP [13], as used for normal avatars. In our case, we developed a variation of XSTEP which we call SignStep, in order to more easily translate to the more intricate movements required for sign execution. The reader may note that the use of SignStep is simply for illustrative purposes, as the design of the system allows for any input notation that can generate the necessary data on the animation queue (via a Java abstract class).

In the next section, we discuss our design in more detail.

3 Design and Implementation

Our avatar animation system consists of three separate modules, namely, a parser, an animator, and a renderer. The system can be incorporated into other systems – for illustrative purposes, we added a user interface module (see Figure 1). The parser module interprets the input notation and communicates with the animator through an animation queue. The renderer module firstly provides the animator with a model to animate (by converting a VRML model into a Java3D model), and secondly sets up a 3D canvas onto which the avatar can be rendered. The animator generates an animated avatar model that is sent to the renderer for display purposes. We now discuss each of the parser, animator and renderer modules in more detail.

The most pertinent feature of the parser module is its handling of the so-called animation queue, with which it communicates with the animator module. The animation queue is quite simply a FIFO linked list of animation actions, ordered from front to back. However, it also has to incorporate concurrent actions, as many signs in sign languages require simultaneous movement of both arms and simultaneous changes in facial expression. To cater for the special case where concurrent movement is required within a sequential movement, we allow for separate nested queues within the primary animation queue. We show an example of a situation where nested queues are needed in Figure 2. The animation queue for a simple animation consisting of four joint rotations is shown. The motion that the queue describes is a rotation of the elbow followed by a rotation of the wrist. If this sequential action is to be executed simultaneously on both sides of the body, two animation queues are created. These two queues contain the sequential actions of a elbow rotation followed by a wrist rotation – the only difference between the two queues is that one refers to the left side of the body while the other refers to the right. To complete the structure, the two queues are then nested inside the primary animation queue as concurrent actions.

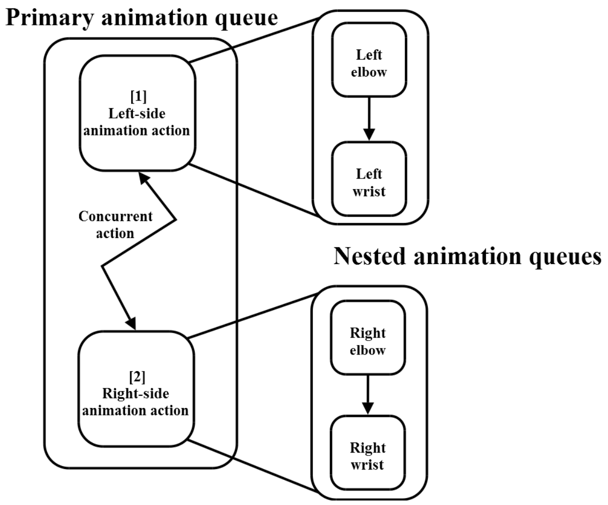
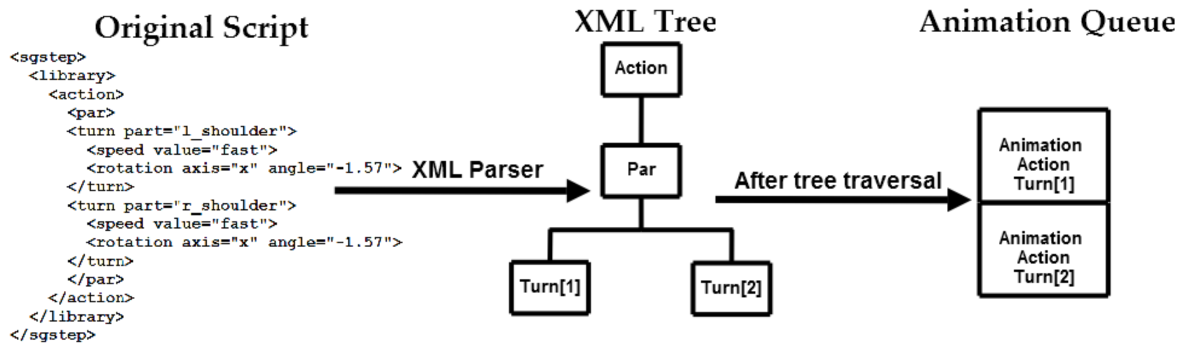


Figure 2. The nested animation queues.

Also relevant to the parser module is the input notation that it expects from outside the system and which should describe the sign to be executed. As mentioned previously, we adapted the XSTEP notation into SignStep. In particular, we discarded features in XSTEP that we deemed not directly relevant to describe signs, and we added finer joint control (for example, in the fingers). Figure 3 illustrates the parsing process, and shows an example of SignStep.

The four most important elements of the SignStep notation are the `seq`, `par`, `turn` and `trans` elements. The `seq` and `par` elements are grouping elements that specifiy whether their nested action elements should be executed concurrently or sequentially. Elements nested inside a `seq` element are executed sequentially and those inside a `par` element are executed concurrently. The two action elements are the `turn` and `trans` elements. The `turn` element represents a joint rotation and must specify the speed, axis of rotation and angle for the desired rotation action. Likewise, the `trans` element specifies a joint translation, and must specify the speed, direction and distance of the



```
<sgstep>
  <library>
    <action>
      <par>
      <turn part="l_shoulder">
        <speed value="fast">
        <rotation axis="x" angle="-1.57">
      </turn>
      <turn part="r_shoulder">
        <speed value="fast">
        <rotation axis="x" angle="-1.57">
      </turn>
      </par>
    </action>
  </library>
</sgstep>
```

Figure 3. The parsing process

translation.

The animator module uses the animation queue constructed by the parser, in combination with the avatar model definition, to build detailed animations to be sent to the renderer for display. These animations are defined in terms of joint data, so that the animator tracks and records joint positions, translations and rotations for the given animation queue.

To apply a given animation queue to a specific avatar model, the Java3D scene graph for that model has to be manipulated by the animator module. In particular, the animator has to insert so-called behavioural nodes into the scene graph for every animation action in the animation queue.

The renderer module provides a platform independent 3D canvas (via Java3D) for the rendering of the animated avatar, and it uses the Xj3D VRML loader package to load an avatar model. During development, we used the then latest version of Xj3D (M10) which currently has some issues related to our project. In particular, the Java3D model is constructed in such a way that the H-Anim joint name references are lost and the joint center of rotations are lost. To compensate for the loss of H-Anim references, we construct a hash map of all the nodes in the model and use the H-Anim joint names as keys to the applicable nodes in the map. The hash map is then sent, together with the avatar model, to the animator module. Compensation for the loss of joint center of rotations is a more difficult problem. Once again, we construct a hash map that is sent to the animator module and we use joint names as keys. However, the joint names here now refer to 3D coordinates instead of the nodes in our model. We fill the hash map by iterating over all the nodes in the original model and extracting the necessary information from each node. A vector of coordinates is then constructed from this information and is inserted into the hash map.

Because it is essential that the signing avatar implementation must be as efficient as possible, we isolated the routines that need optimisation in the system. On average, the system spends 71% of its CPU time in the `loadfile` method of the renderer module. We also identified the data structures that are responsible for the largest memory usage in our system. In this case, the `XObject` object that forms part of the XPath package is mostly to blame. The XPath package is used to query the XML tree that is built from our input script in the parser module.

To reduce the CPU usage of our system, we are currently investigating the first stable production release of the Xj3D loader which should, to our understanding, address this issue (this version of Xj3D was released in April 2006). The second optimisation that will be of benefit is the memory usage of the XML parser in the parser module. We originally decided to use a DOM parser, since the tree structure that the DOM parser builds significantly simplifies the construction of our animation queue. However, it is exactly this tree structure that is responsible for our memory footprint. The other option is to use a SAX parser, but that will complicate the construction of the animation queue – we are currently investigating whether the use of a SAX parser will be worthwhile in the view of the additional computational effort.

4 Results

Our signing avatar animation system has been designed to be as generic as possible for as many applications as possible. We tested it extensively with a (typical) user interface, and we illustrate the results below.

Consider the English sentence *I don't know* and its South African Sign Language equivalent. Figure 4 shows a screenshot from a cartoon-based avatar performing the sign, while Figure 5 shows a screenshot of the Nancy avatar [14] performing the same sign. Both avatars were given the script shown in Figure 6 as input. Note that the slight discontinuities at the wrists of the Nancy avatar can be improved by a better avatar definition, and is not due to faulty animations.

Our system can successfully perform animations, such as *I don't know* illustrated above, without any programming intervention, and its design in this sense is more than satisfactory. There are, however, in many sign languages, certain signs that are difficult to describe generically from a physiological point of view. One example is a splayed left hand and a pointed finger from the right

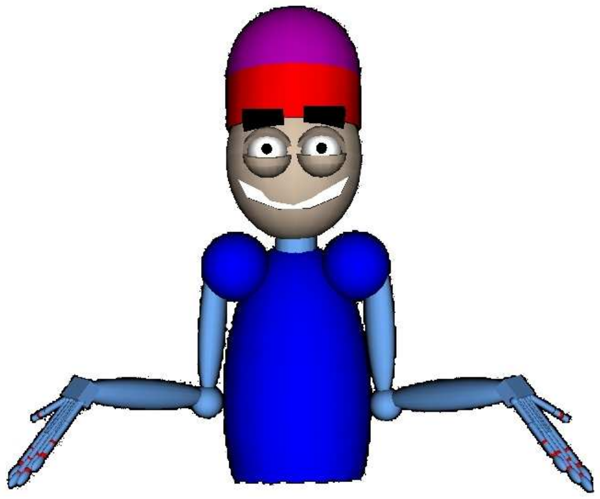


Figure 4. Cartoon character signing *I don’t know*.

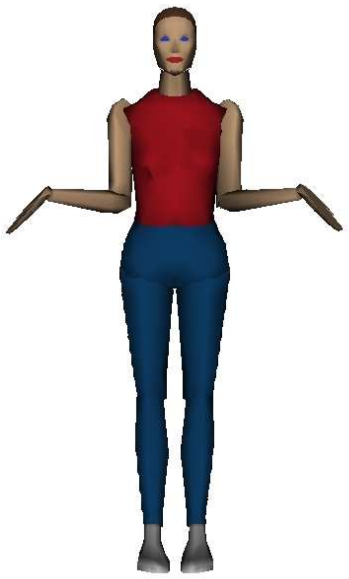


Figure 5. Nancy humanoid signing *I don’t know*.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE sgstep SYSTEM "SGSTEP.dtd">
<sgstep> <library>
<action name="shoulder_rotate"> <par>
 <trans part="l_shoulder">
   <speed value="very_fast"/>
   <dir axis="y" distance="0.032"/></trans>
 <trans part="r_shoulder">
   <speed value="very_fast"/>
   <dir axis="y" distance="0.032"/></trans>
 <turn part="l_elbow">
   <speed value="very_fast"/>
   <rotation axis="x" angle="-1.7"/></turn>
 <turn part="r_elbow">
   <speed value="very_fast"/>
   <rotation axis="x" angle="-1.7"/></turn>
 <turn part="l_shoulder">
   <speed value="very_fast"/>
   <rotation axis="y" angle="0.78"/></turn>
 <turn part="r_shoulder">
   <speed value="very_fast"/>
   <rotation axis="y" angle="-0.78"/></turn>
 <turn part="l_elbow">
   <speed value="very_fast"/>
   <rotation axis="y" angle="1.5"/></turn>
 <turn part="r_elbow">
   <speed value="very_fast"/>
   <rotation axis="y" angle="-1.5"/></turn>
 <turn part="l_wrist">
   <speed value="very_fast"/>
   <rotation axis="z" angle="1.0"/></turn>
 <turn part="r_wrist">
   <speed value="very_fast"/>
   <rotation axis="z" angle="-1.0"/></turn>
</par> </action> </library> </sgstep>}
```

Figure 6. SignStep script for *I don’t know*.

hand touching the fingers of the left hand (as if counting on the fingers). Since finger length is related to but not specifically defined by the overall size of the avatar (a tall person may indeed have short fingers), a given SignStep script may cause the finger tips to miss contact. This can be remedied manually by adapting the SignStep script by hand, but clearly this is an issue where no system can be truly generic.

The reader may note that collisions are fully avoided by a correct SignStep script. In typical sign language transcriptions, the signs are formed by transcribing them directly from videos of human signers. Since human signers sequentialize signs to prevent collisions, these transcriptions avoid collisions and hence a typical SignStep script file will mimic this situation.

5 Future Work

We are still working on extensions to our system, as well as tools to simplify its use.

One of the more intricate extensions involve the modelling of facial expressions. This is particularly difficult, as H-Anim has limited facial joints. The facial expressions for sign languages require control over eyebrows, eyelids, eye gaze, lips, tongue, mouth and also cheek movements. One possibility for our system is to extend H-Anim to allow for more facial joints – this however influences the genericity of the system.

Another useful extention to our system would be a web interface. Since the system was developed entirely in Java, we can convert our current GUI interface into a Java Web applet with little extra programming. However, the overhead of the Java applet class loader and applet security would cause unacceptable loss in frame rate, and this issue would need careful investigation.

As far as tools are concerned, it may be noted that at this stage the production of a SignStep script file is manual and tedious. We are therefore currently developing a sign editor tool [15] which would rectify this problem. The sign editor will provide visual construction of a sign and then produce a SignStep script file as output. Moreover, the editor will allow for partial key frame sequences and intelligent joining of different files to produce phrases instead of just single words.

6 Conclusion

We discussed the design and implementation of a generic signing avatar. The advantages to our system include the capability of using it with any reasonable H-Anim compliant avatar, the capability of coupling it easily to other applications, and its performance.

In summary, at this stage we have a working signing avatar with restricted facial movements. The animations are generic, and have been demonstrated on both humanoid and cartoon avatar models. Our system is easy to install, runs on both Linux and Windows environments, and are distributed free for personal and academic use.

Acknowledgements

This research was supported by NRF grant number 2053436.

References

[1] S. Yeates, E.-J. Holden, and R. Owens, An Animated Auslan Tuition System, *Machine Graphics and Vision International Journal 12(2)*, 2003, 203–214.

[2] S-E. Fotinea, E. Efthimiou, K. Karpouzis and C. Caridakis, Dynamic GSL Synthesis to Support Access to e-content, *Proceedings of the 3rd International Conference on Universal Access in Human-Computer Interaction*, Las Vegas, USA, 2005.

[3] N. Suszczanska, P. Szmal and F. Jaroslaw, Translating Polish Text into Sign Language in the TGT System, *Proceedings of the 20th IASTED International Multi-Conference on Applied Informatics*, Innsbruck, Austria, 2002, 282–287.

[4] L. van Zijl and D. Barker, The South African Sign Language Machine Translation Project, *Proceedings of Afrigraph 2003*, Cape Town, South Africa, 2003, 49–52.

[5] R. Kennaway, Synthetic Animation of Deaf Signing Gestures, *Lecture Notes in Artificial Intelligence 2298*, 2003, 146–157.

[6] J. Toro, K. Alkoby, R.Carter, J. Christopher, B. Craft, M. Davidson, D. Hinkle, G. Lancaster, A. Morris, J. McDonald, E. Sedgwick, and R. Wolfe, *A Graphical Environment for Transcription and Display of American Sign Language*, Information 4(4), 2001, 533–539.

[7] VCom3D, `http://www.vcom3d.com`.

[8] Humanoid Animation Working Group, `http://h-anim.org`.

[9] Xj3D, `http://www.xj3d.org`.

[10] Blaxxun Contact, `http://www.blaxxun.com`.

[11] X. Yang, D. Petriu, T. Whalen and E. Petriu, Hierarchical Animation Control of Avatars in 3D Environments, *IEEE Transactions for Instrumentation and Measurement 54*, 2005, 1333—1341.

[12] Sun Microsystems, The Java3D API, `http://java.sun.com/products/java-media/3D/`.

[13] Z. Huang, A. Eliëns and C. Visser, Implemenation of a Scripting Language for VRML/X3D-based Embodied Agents, *Proceedings of the Web3D 2003 Symposium*, 2003, 91–100.

[14] C. Ballreich, The Nancy Avatar, `http://www.ballreich.net/vrml/h-anim/nancy_h-anim.wrl`.

[15] D. Potgieter, A Sign Editor Tool. `http://www.cs.sun.ac.za/~dpotgieter/`